DCA0204, Módulo 3
Filas e Busca em Largura (BFS)

Daniel Aloise

baseado em slides do prof. Leo Liberti, École Polytechnique, França

Sumário

- Motivação
- Filas
- Busca em Largura (BFS)
- Implementação

Motivação

Um exemplo do 1o. mundo

	A
1	h:00
2	h:10
3	h:30

	B
1	h:00
4	h:20
5	h:40

	C
2	h:10
3	h:20
5	h:30

	D
4	h:20
5	h:40
6	h:50

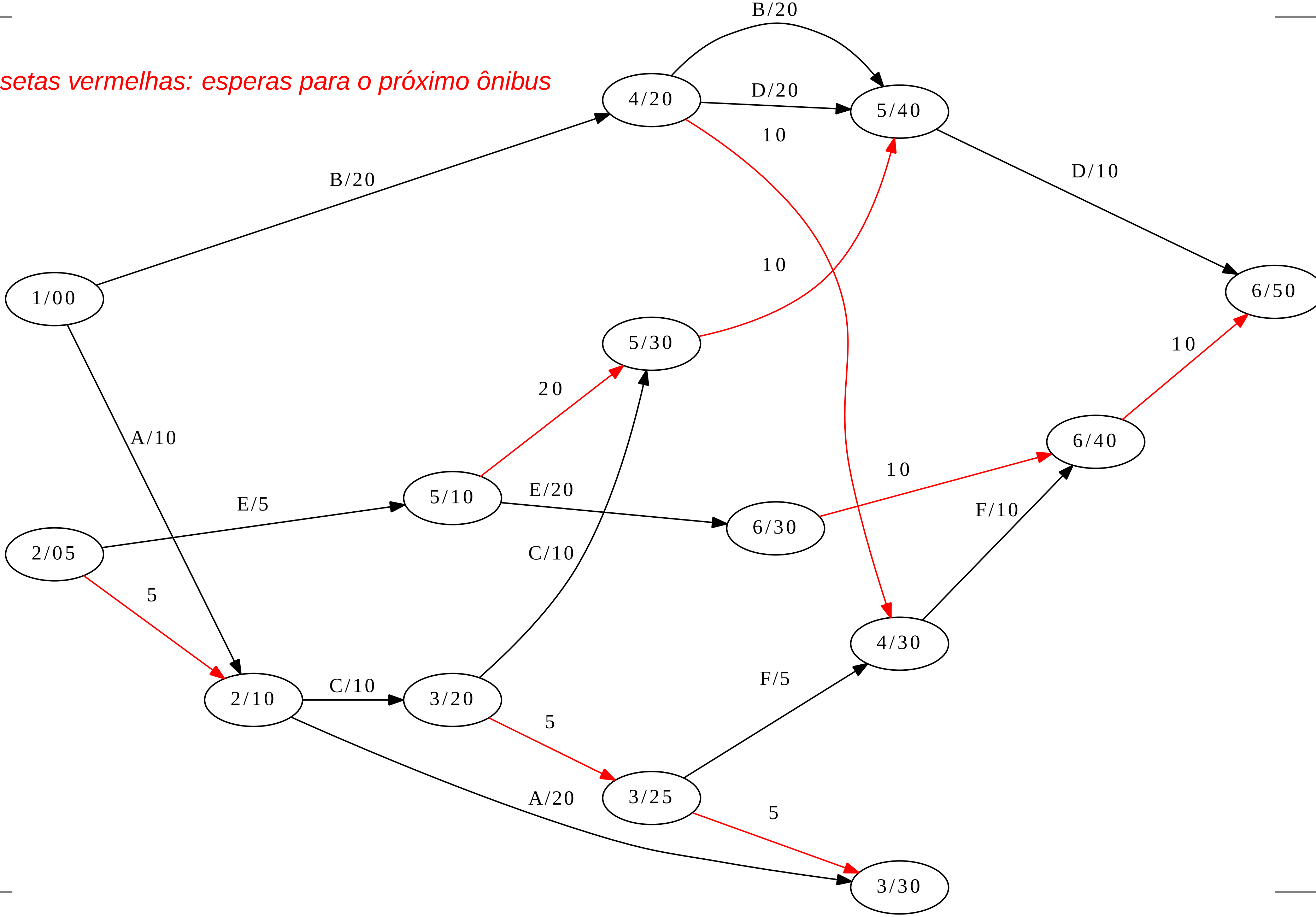
	E
2	h:05
5	h:10
6	h:30

	F
3	h:25
4	h:30
6	h:40

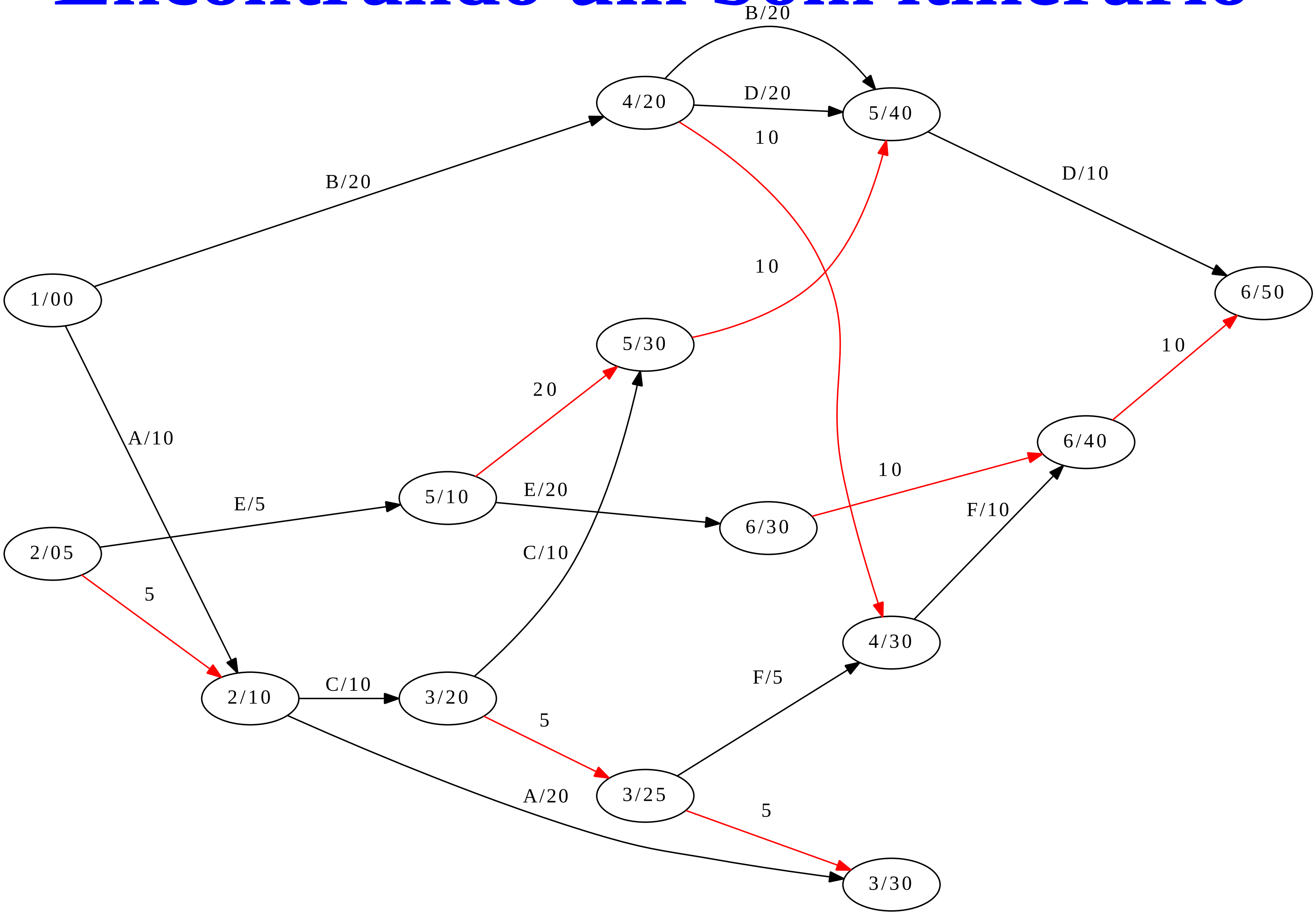
Encontre um itinerário de 1 para 6, saindo às h:00?

O grafo de eventos

setas vermelhas: esperas para o próximo ônibus

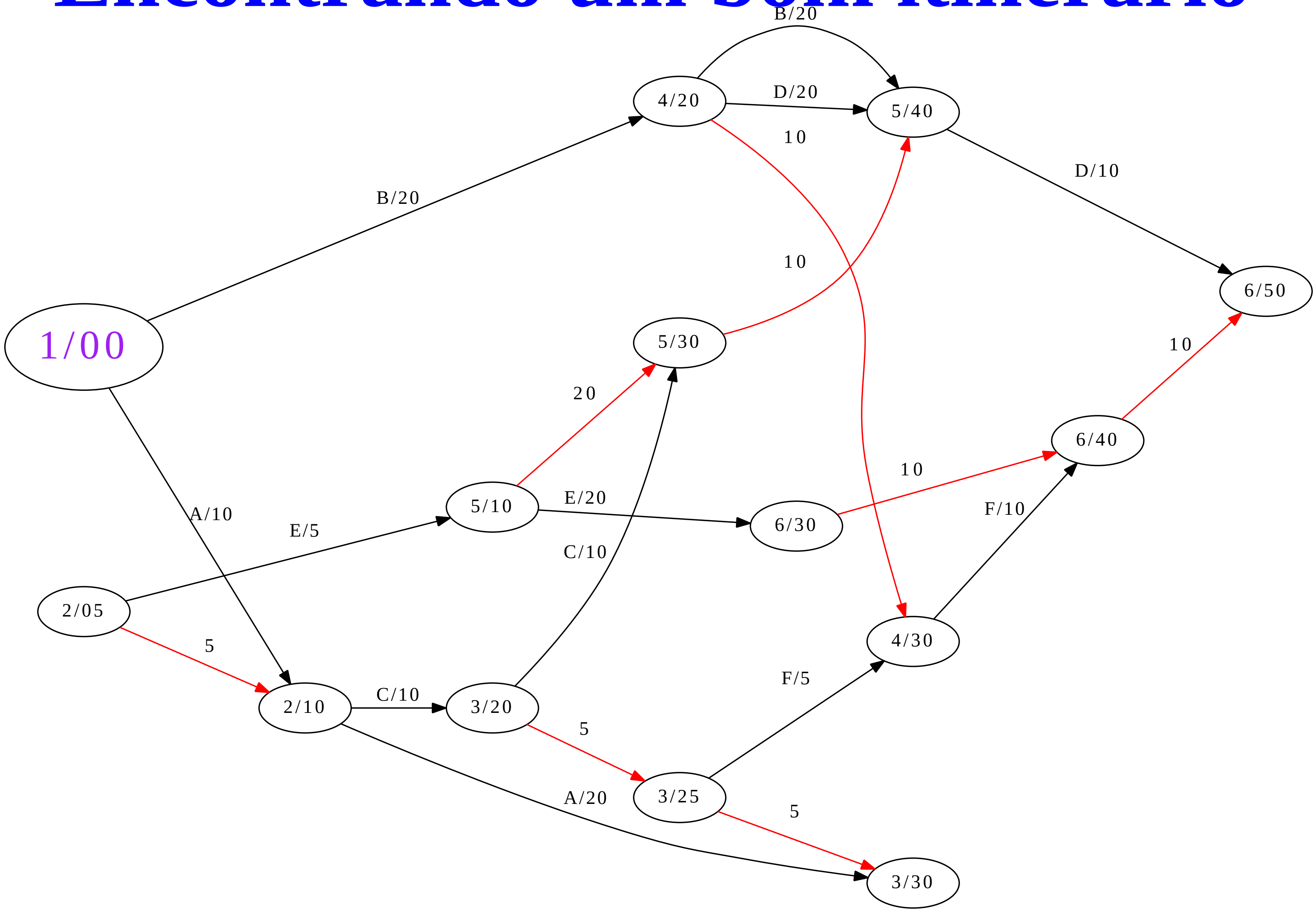


Encontrando um bom itinerário



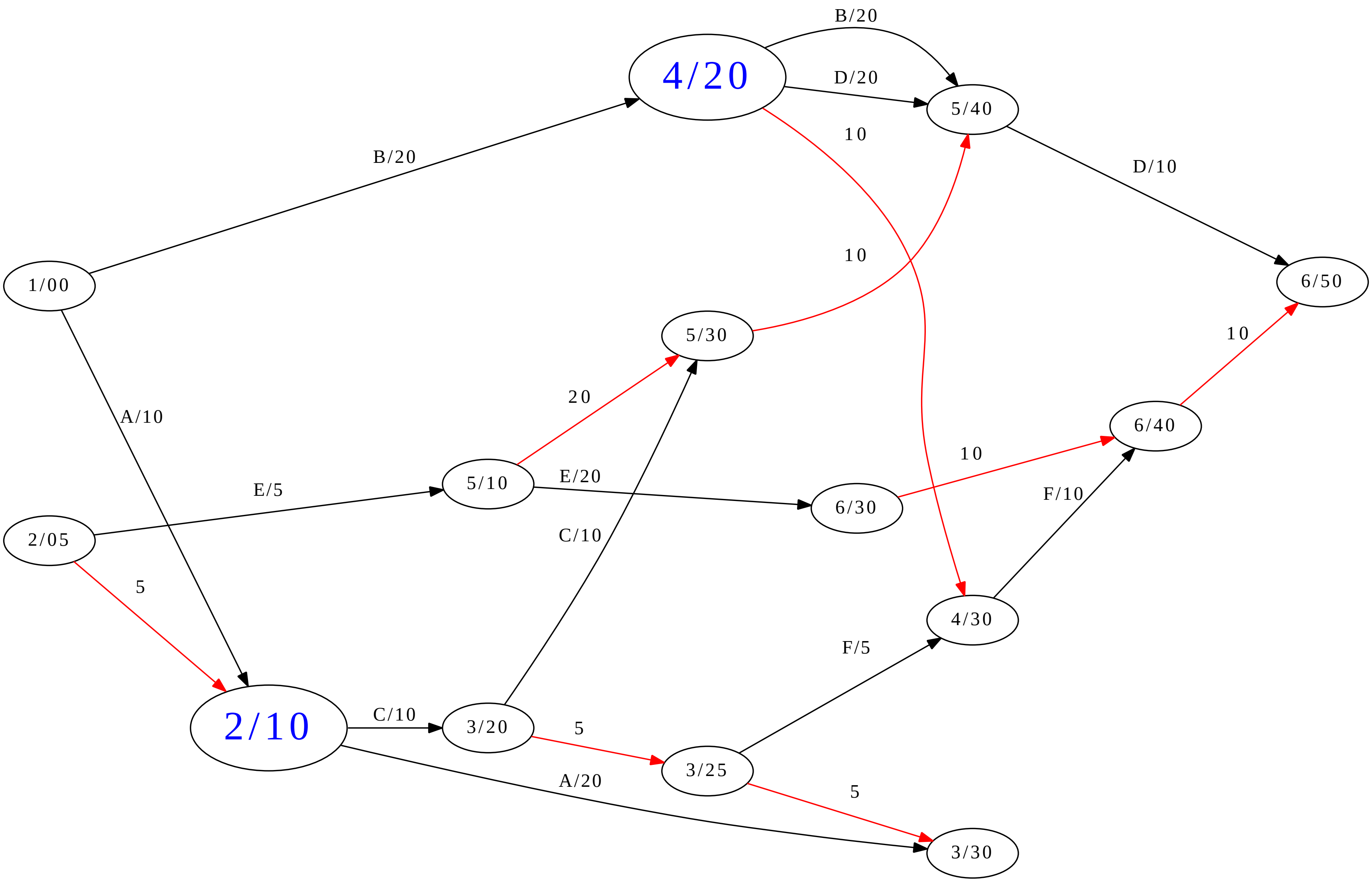
1/00

Encontrando um bom itinerário



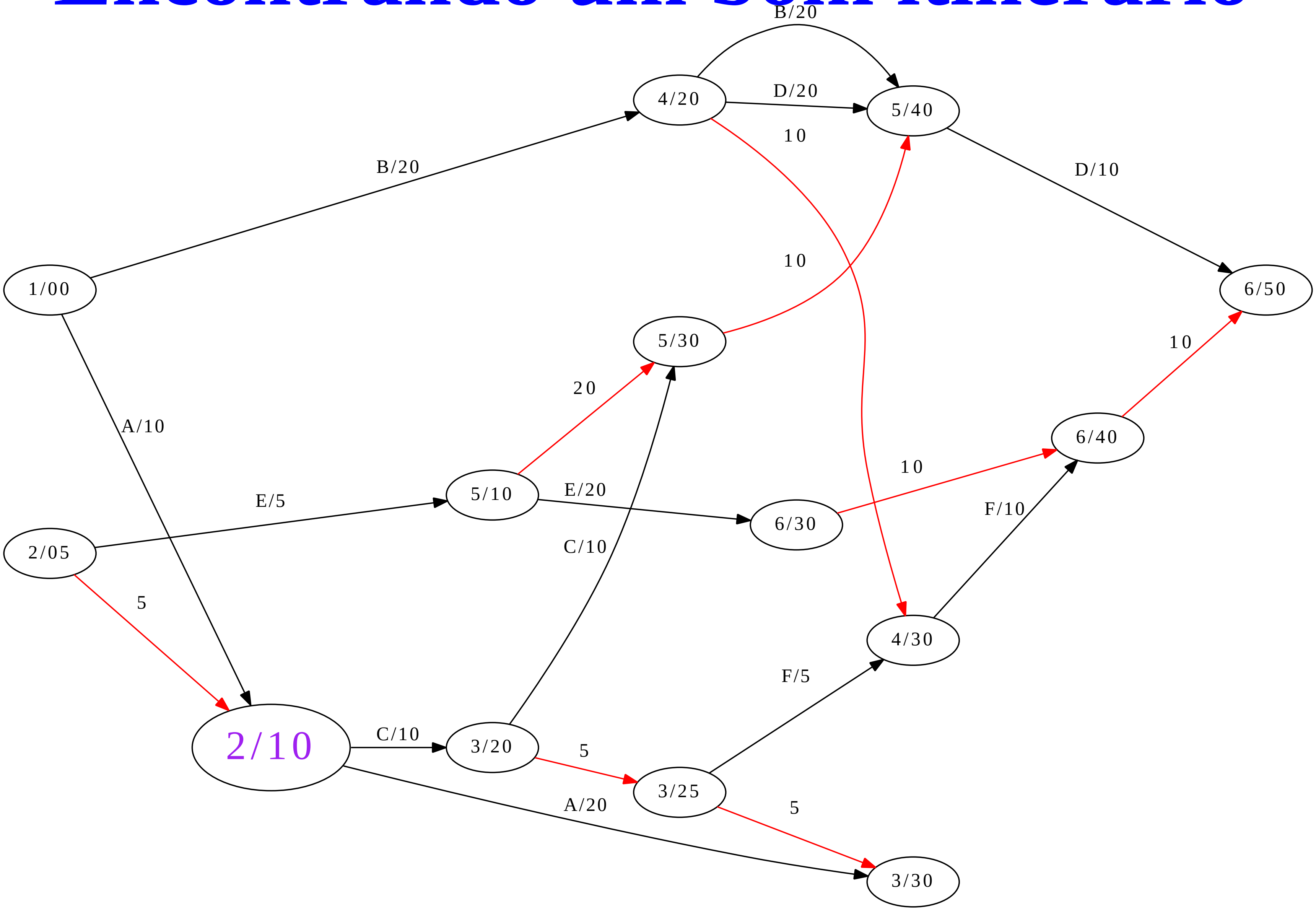
1/00 ←

Encontrando um bom itinerário



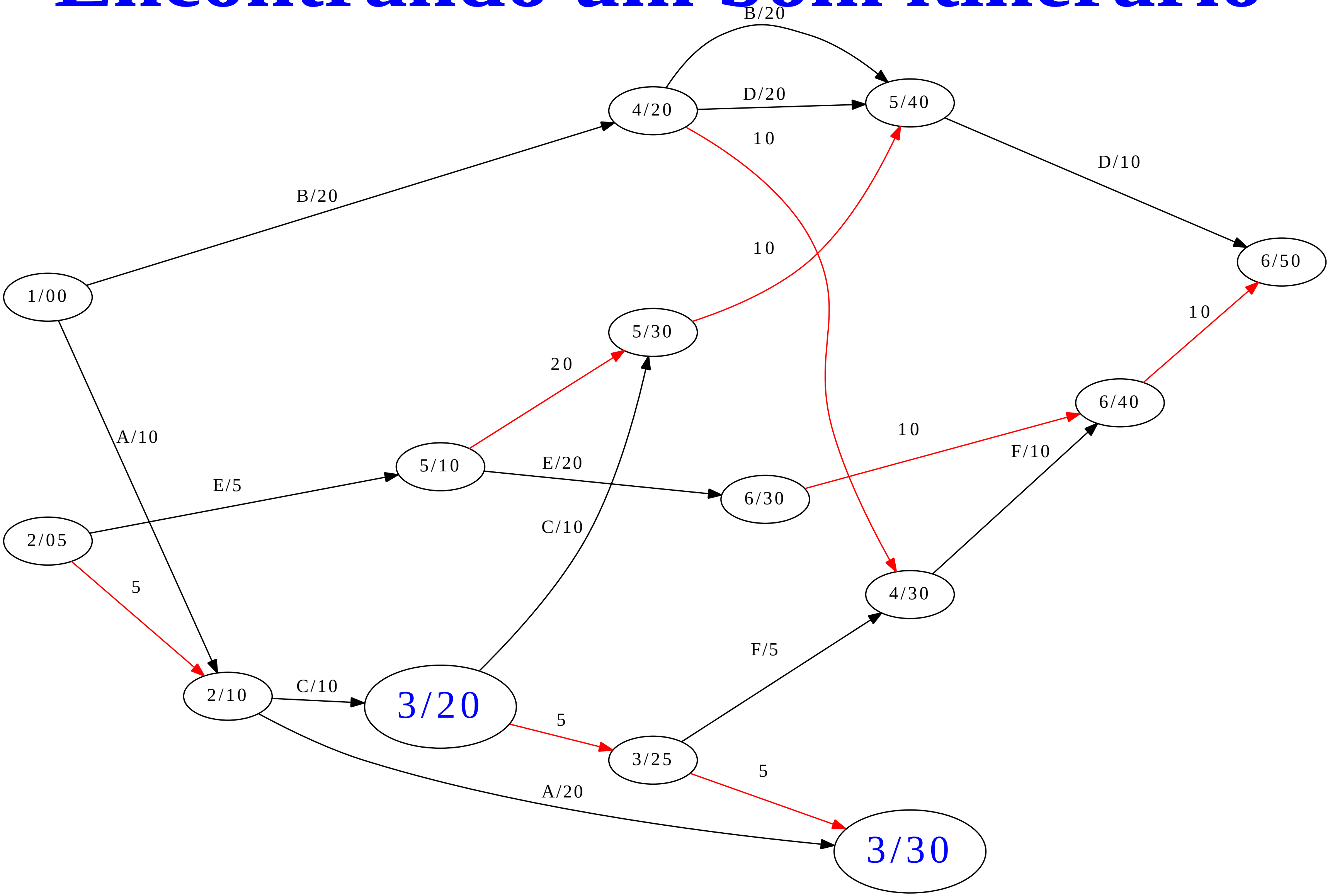
2/10	4/20

Encontrando um bom itinerário



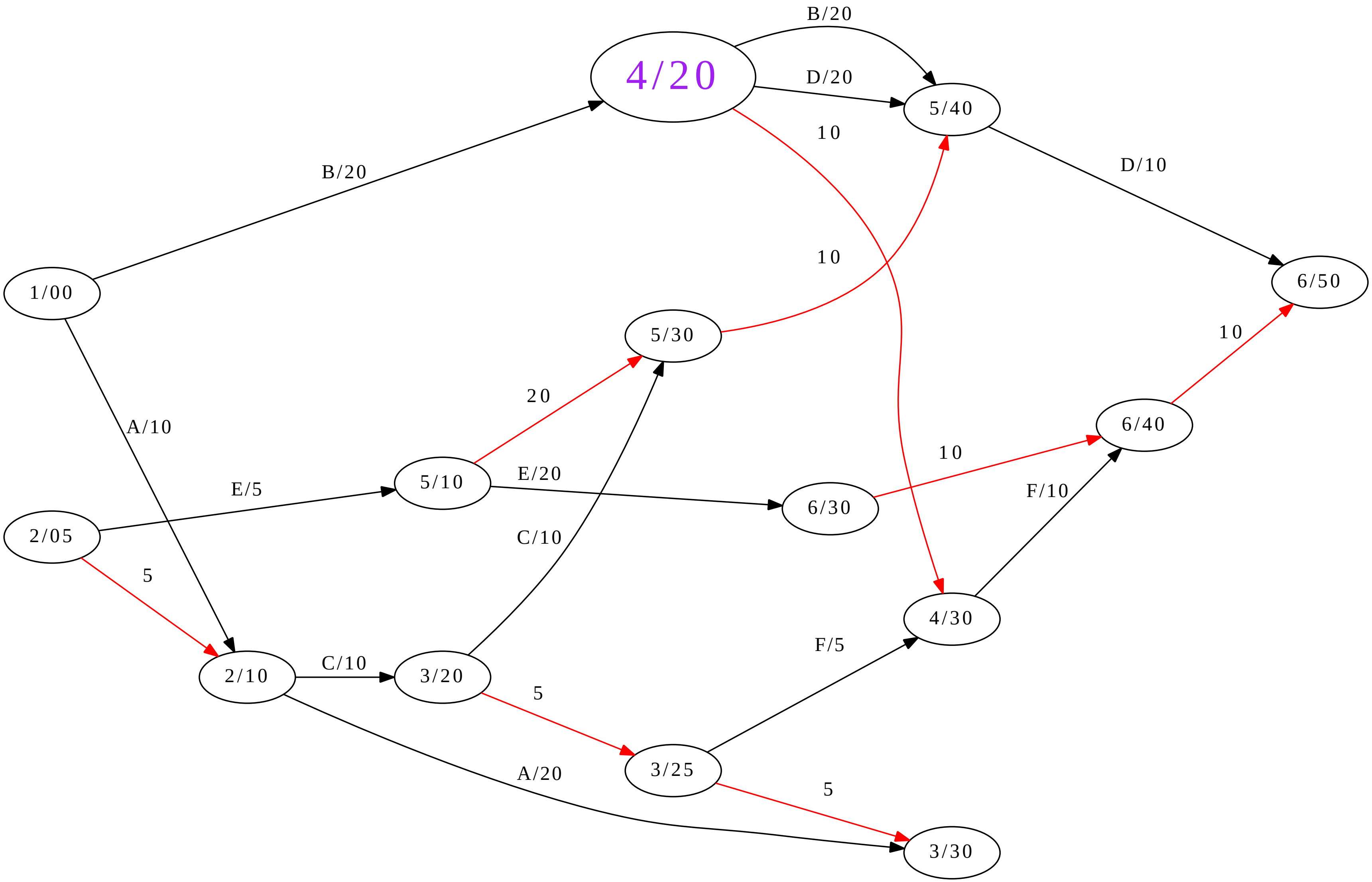
2/10 ← $\boxed{4/20}$

Encontrando um bom itinerário



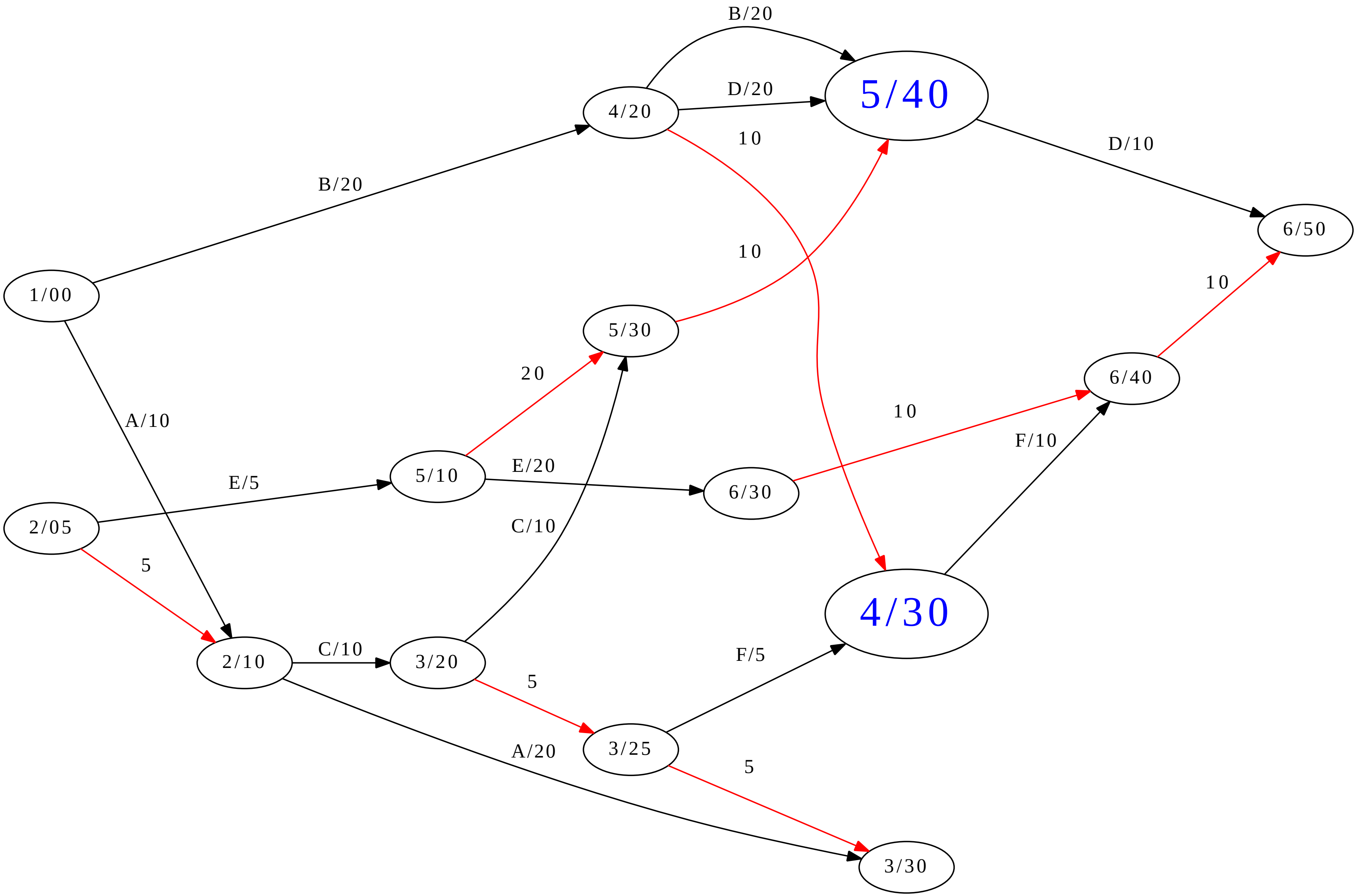
4/20	3/20	3/30

Encontrando um bom itinerário



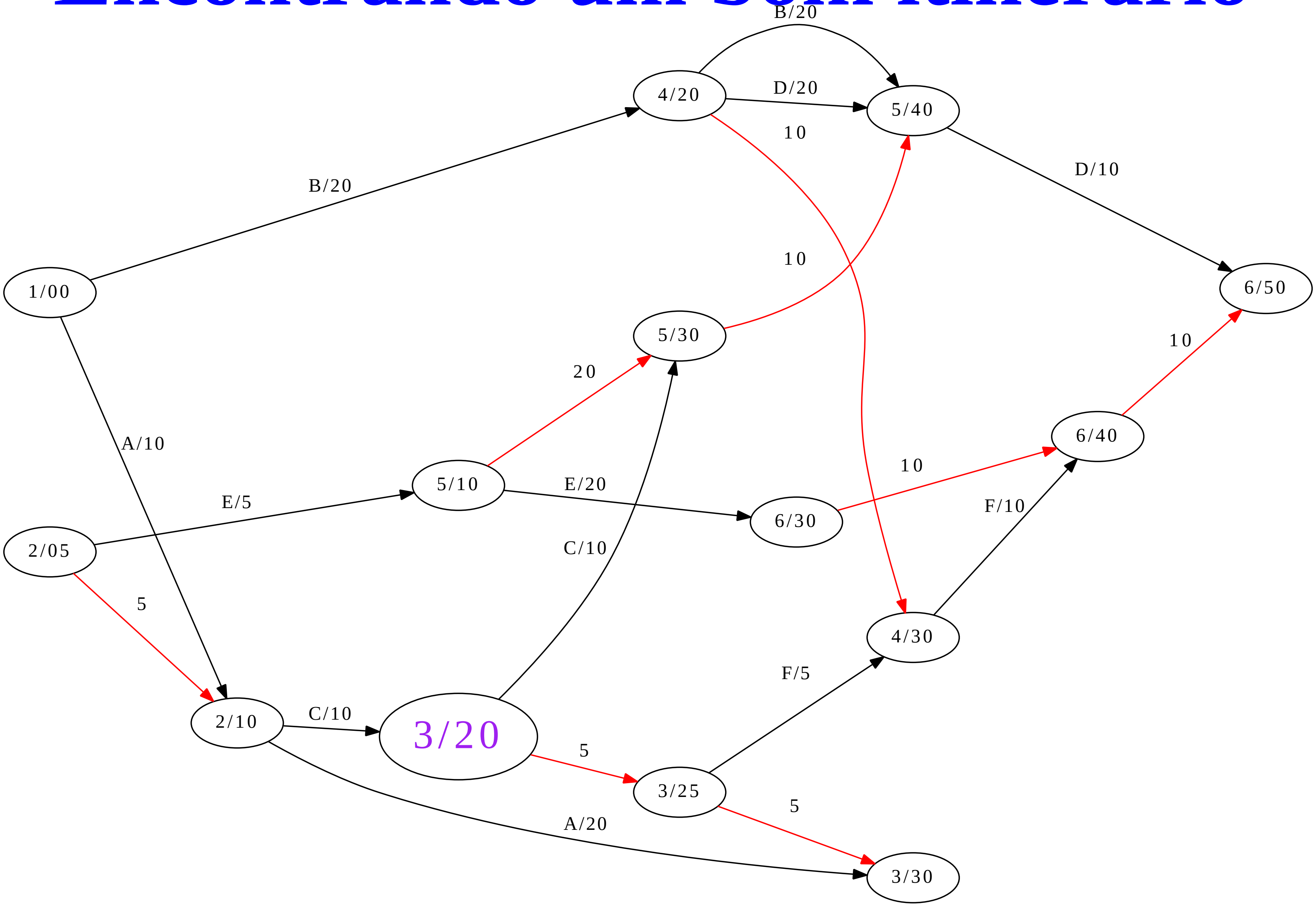
4/20 ← | 3/20 | 3/30 |

Encontrando um bom itinerário



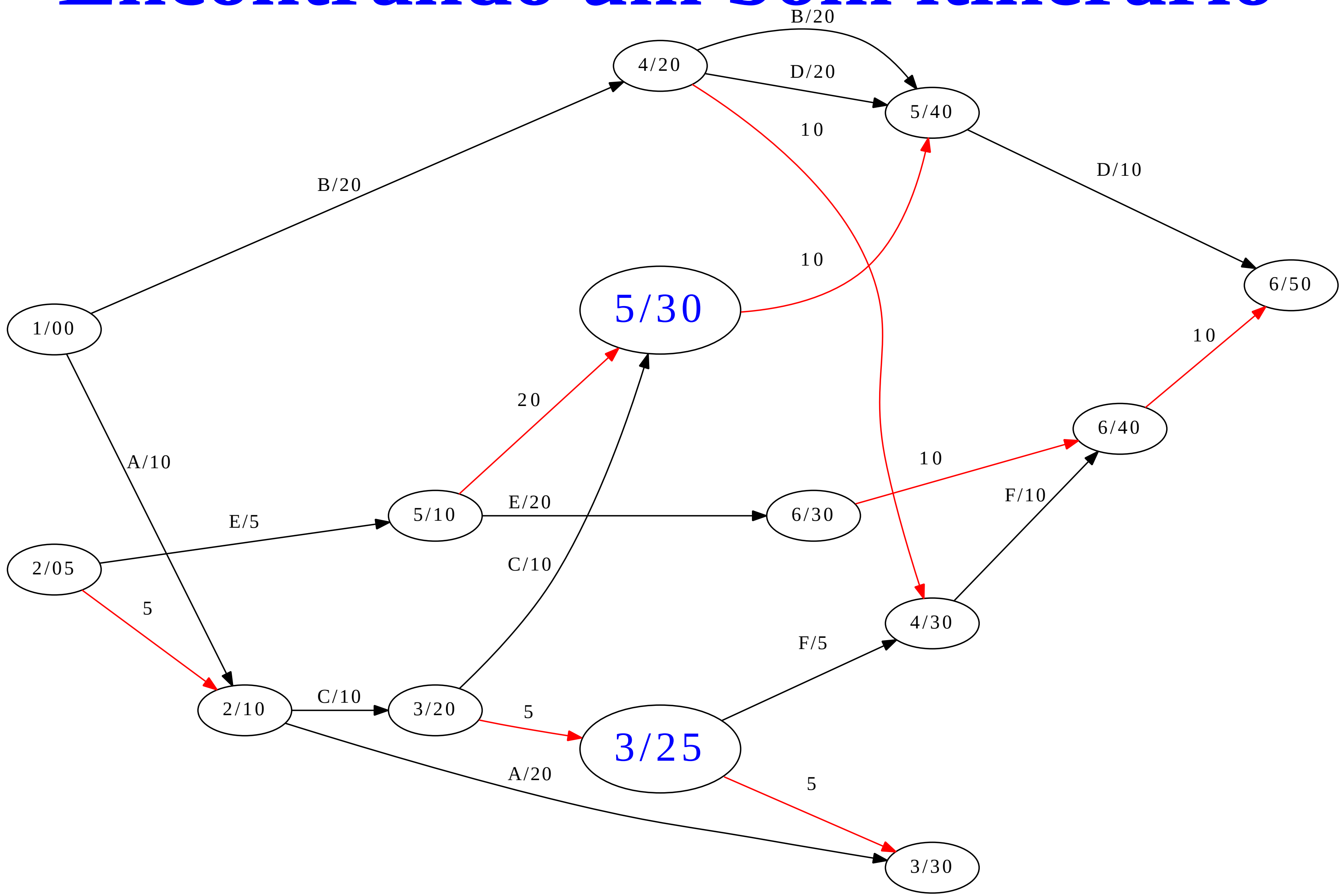
3/20	3/30	4/30	5/40

Encontrando um bom itinerário



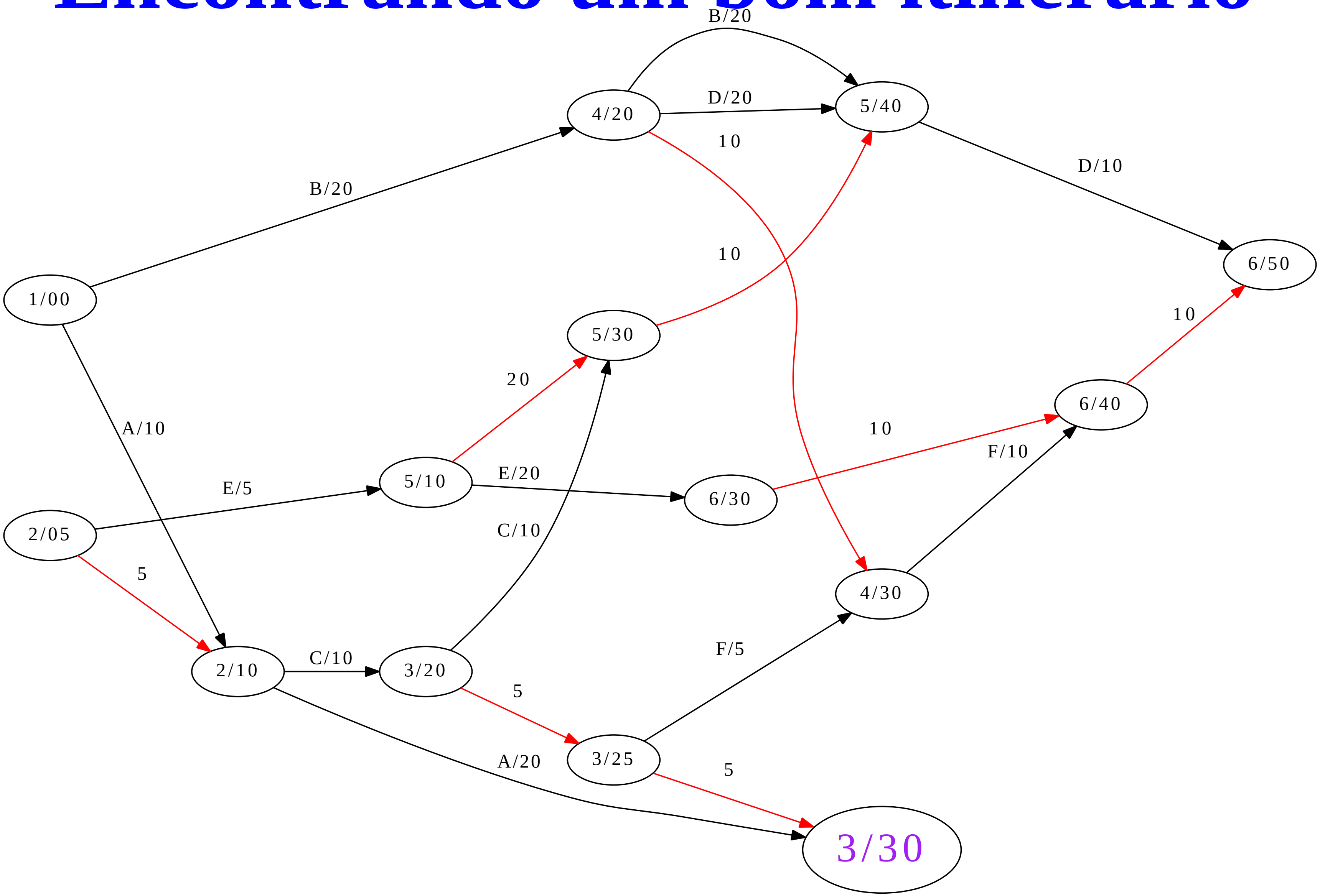
3/20 ← | 3/30 | 4/30 | 5/40 |

Encontrando um bom itinerário



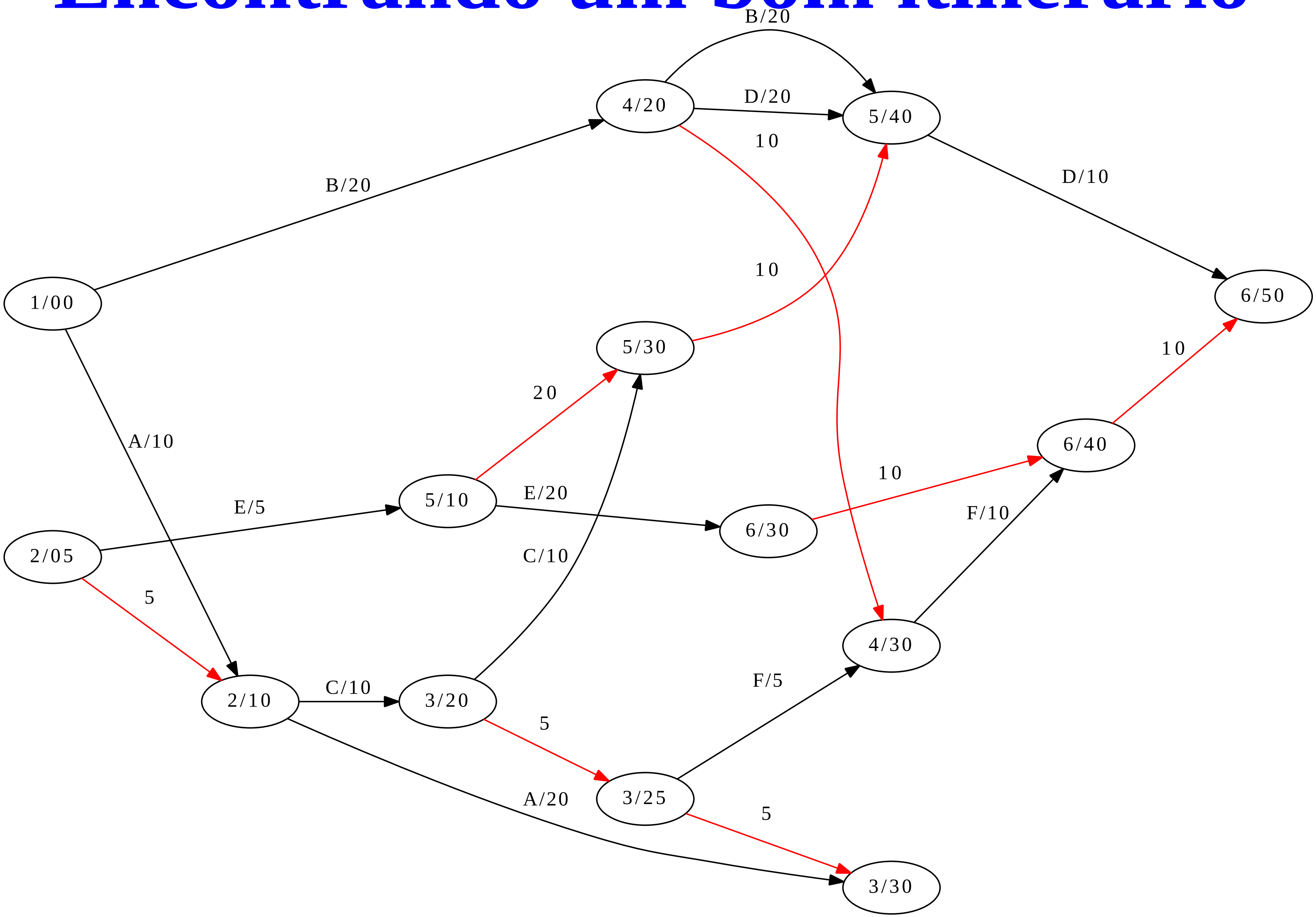
3/30	4/30	5/40	3/25	5/30

Encontrando um bom itinerário



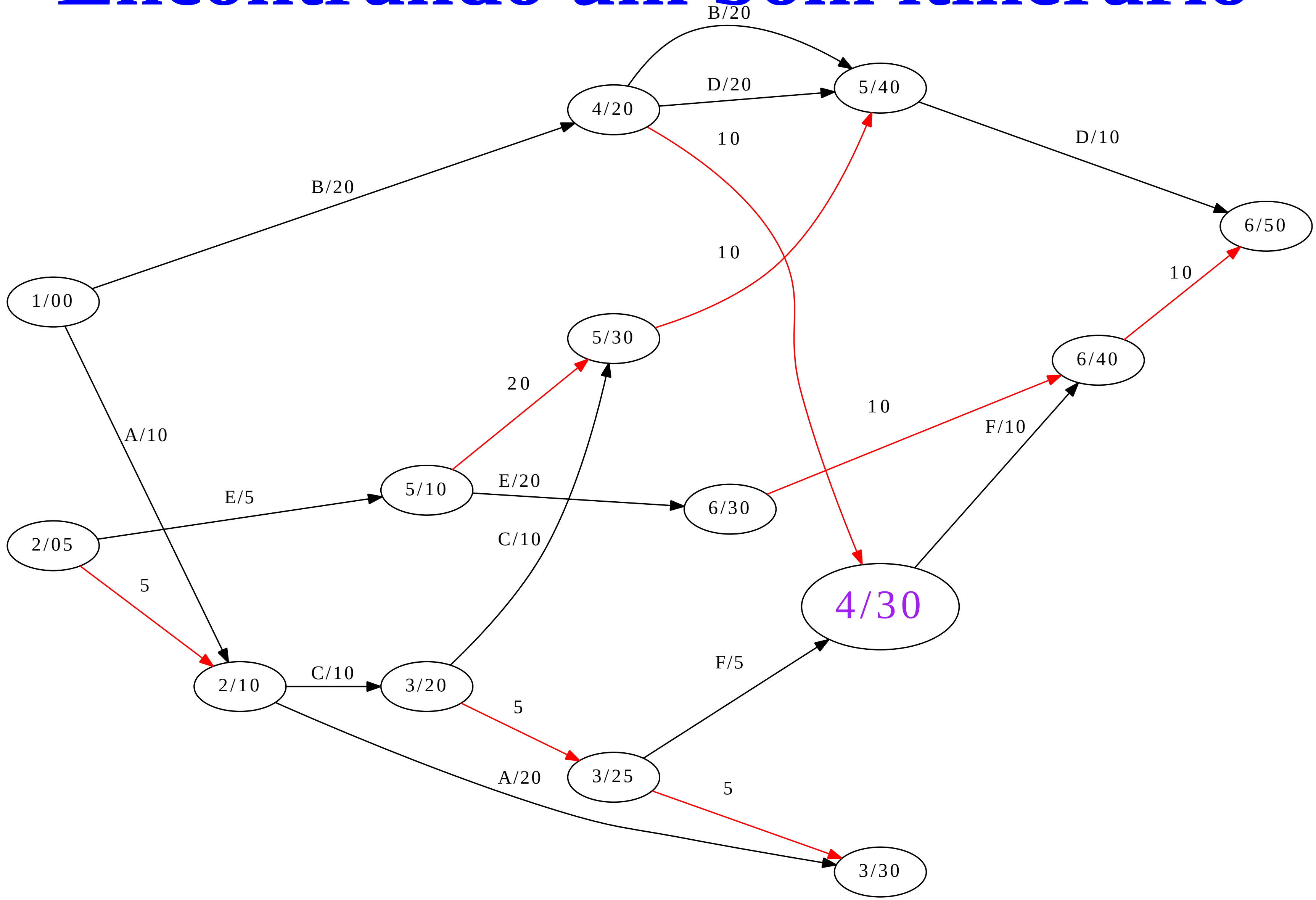
3/30 ← | 4/30 | 5/40 | 3/25 | 5/30 |

Encontrando um bom itinerário



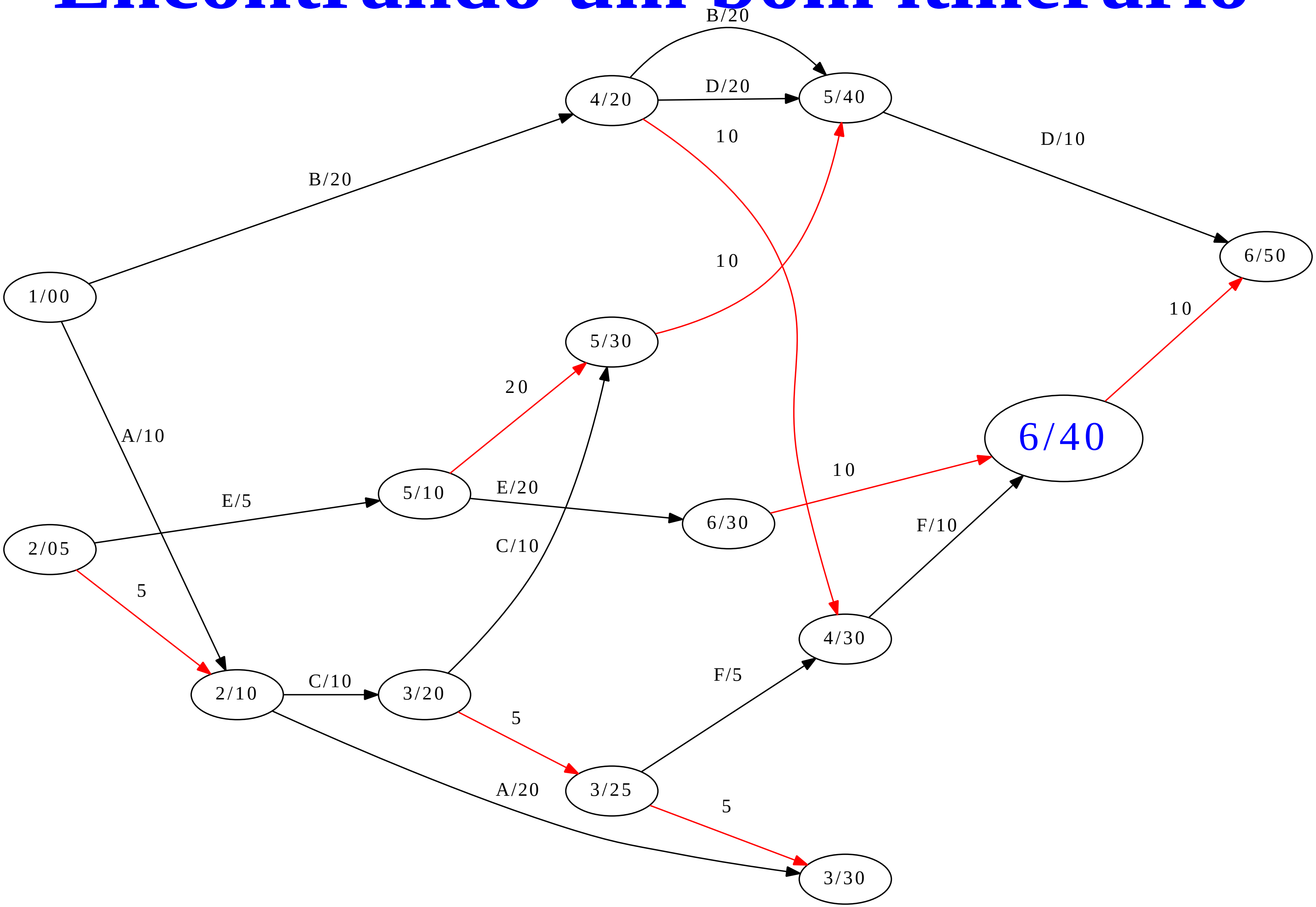
4/30	5/40	3/25	5/30

Encontrando um bom itinerário



4/30 ← | 5/40 | 3/25 | 5/30 |

Encontrando um bom itinerário



5/40	3/25	5/30	6/40

itinerário encontrado 1$\longrightarrow$6 chegando às h:40

Recuperando o caminho

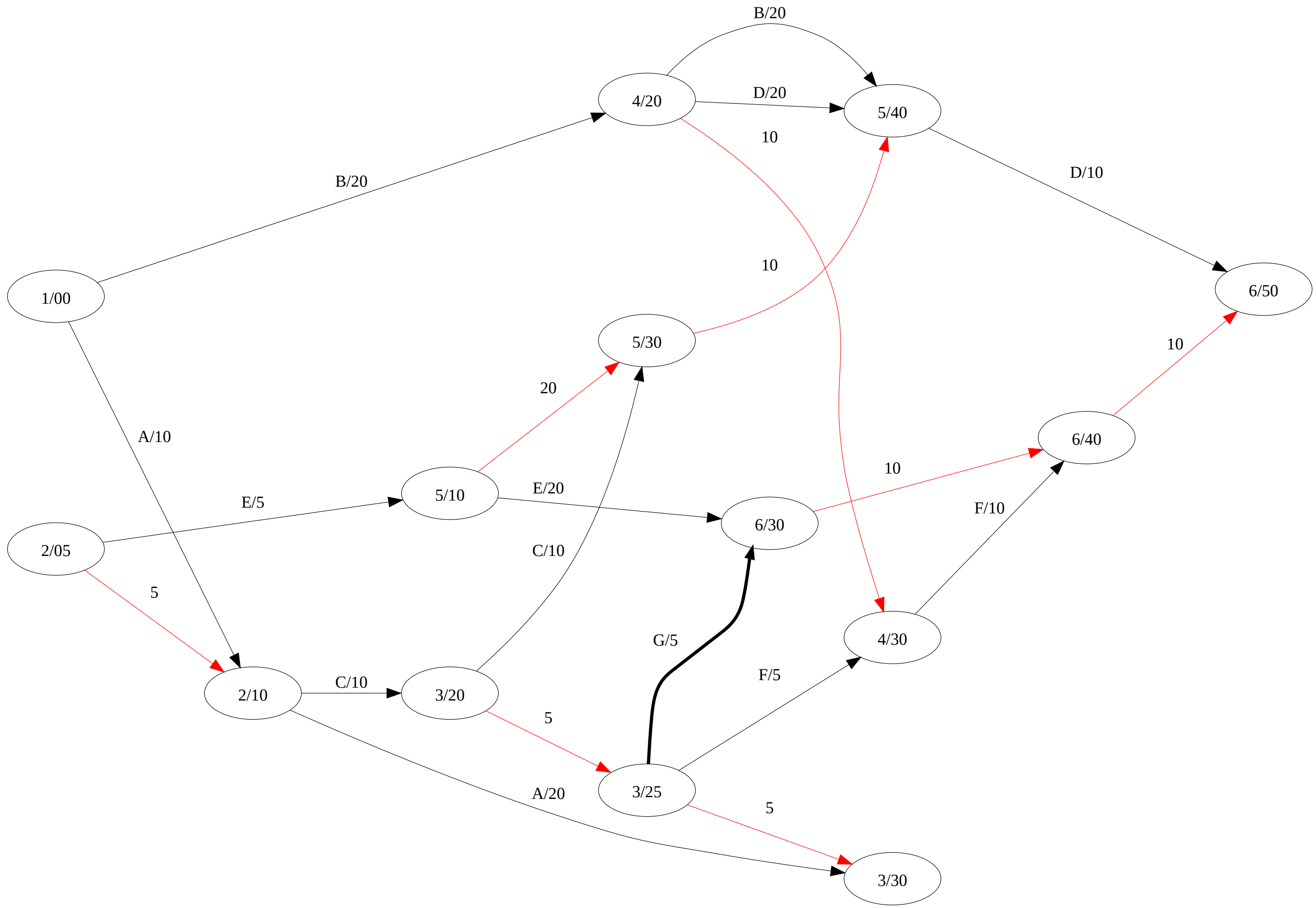
- O método anterior nos fornece apenas a duração, não o caminho realizado
- *A cada iteração, armazene os nós na fila com seus respectivos predecessores*

pred	nó
-	1/00
1/00	2/10
1/00	4/20
2/10	3/20
2/10	3/30
4/20	4/30
4/30	6/40

- Agora recupere o caminho de trás para frente: 6/40 $\rightarrow$ 4/30 $\rightarrow$ 4/20 $\rightarrow$ 1/00

Não é o mais rápido

Suponha que exista um ônibus G com timetable 3/25 $\rightarrow$ 6/30



3/25 ainda estará na fila (5/40 | 3/25 | 5/30 | 6/40) após a terminação , não chega a 6/30

O que nós encontramos?

- A fim de se encontrar o itinerário *mais rápido*, precisamos modificar a condição de parada

ao invés de parar quando o nó de chegada é encontrado,

Seja τ^* o tempo de chegada da melhor solução até agora

Seja τ' o tempo mínimo dos nós na fila

if $(\tau' \geq \tau^*)$ **then**

termine o programa

endif

O que nós encontramos?

- A fim de se encontrar o itinerário *mais rápido*, precisamos modificar a condição de parada

  ao invés de parar quando o nó de chegada é encontrado,

  Seja τ^* o tempo de chegada da melhor solução até agora

  Seja τ' o tempo mínimo dos nós na fila

  if $(\tau' \geq \tau^*)$ **then**

  termine o programa

  endif

- Quais são as propriedades do nosso itinerário?

O que nós encontramos?

- A fim de se encontrar o itinerário *mais rápido*, precisamos modificar a condição de parada

ao invés de parar quando o nó de chegada é encontrado,

Seja τ^* o tempo de chegada da melhor solução até agora

Seja τ' o tempo mínimo dos nós na fila

if $(\tau' \geq \tau^*)$ **then**

termine o programa

endif

- Quais são as propriedades do nosso itinerário?

- Nós encontramos o **itinerário com o menor número de trechos**

onde "ônibus, espera, ônibus" conta como dois trechos, não um.

O que nós encontramos?

- A fim de se encontrar o itinerário *mais rápido*, precisamos modificar a condição de parada

ao invés de parar quando o nó de chegada é encontrado,

Seja τ^* o tempo de chegada da melhor solução até agora

Seja τ' o tempo mínimo dos nós na fila

if $(\tau' \geq \tau^*)$ **then**

termine o programa

endif

- Quais são as propriedades do nosso itinerário?
- Nós encontramos o **itinerário com o menor número de trechos**

onde "ônibus, espera, ônibus" conta como dois trechos, não um.

- A fim de provar esta afirmação, precisamos formalizar o nosso método através de um algoritmo

O que nós encontramos?

- A fim de se encontrar o itinerário *mais rápido*, precisamos modificar a condição de parada

  ao invés de parar quando o nó de chegada é encontrado,

  Seja τ^* o tempo de chegada da melhor solução até agora

  Seja τ' o tempo mínimo dos nós na fila

  if $(\tau' \geq \tau^*)$ **then**

  termine o programa

  endif

- Quais são as propriedades do nosso itinerário?
- Nós encontramos o **itinerário com o menor número de trechos**

  onde "ônibus, espera, ônibus" conta como dois trechos, não um.

- A fim de provar esta afirmação, precisamos formalizar o nosso método através de um algoritmo
- Portanto, precisamos descrever nossa "fila" como uma estrutura de dados.

Filas

Operações de uma fila

- No nosso exemplo, nós precisávamos que nossa “fila” fosse capaz de realizar as seguintes operações:
  - **inserir** *um elemento no* **fim** *da fila*
  - **recuperar e deletar** *o elemento no* **começo** *da fila*
  - *testar se a fila* **está vazia**
  - *encontrar o* **tamanho** *da fila*

Operações de uma fila

- No nosso exemplo, nós precisávamos que nossa “fila” fosse capaz de realizar as seguintes operações:
  - **inserir** *um elemento no* **fim** *da fila*
  - **recuperar e deletar** *o elemento no* **começo** *da fila*
  - *testar se a fila* **está vazia**
  - *encontrar o* **tamanho** *da fila*
- Uma vez que estas operações são repetidas com frequência, precisamos que elas sejam $O(1)$

Operações de uma fila

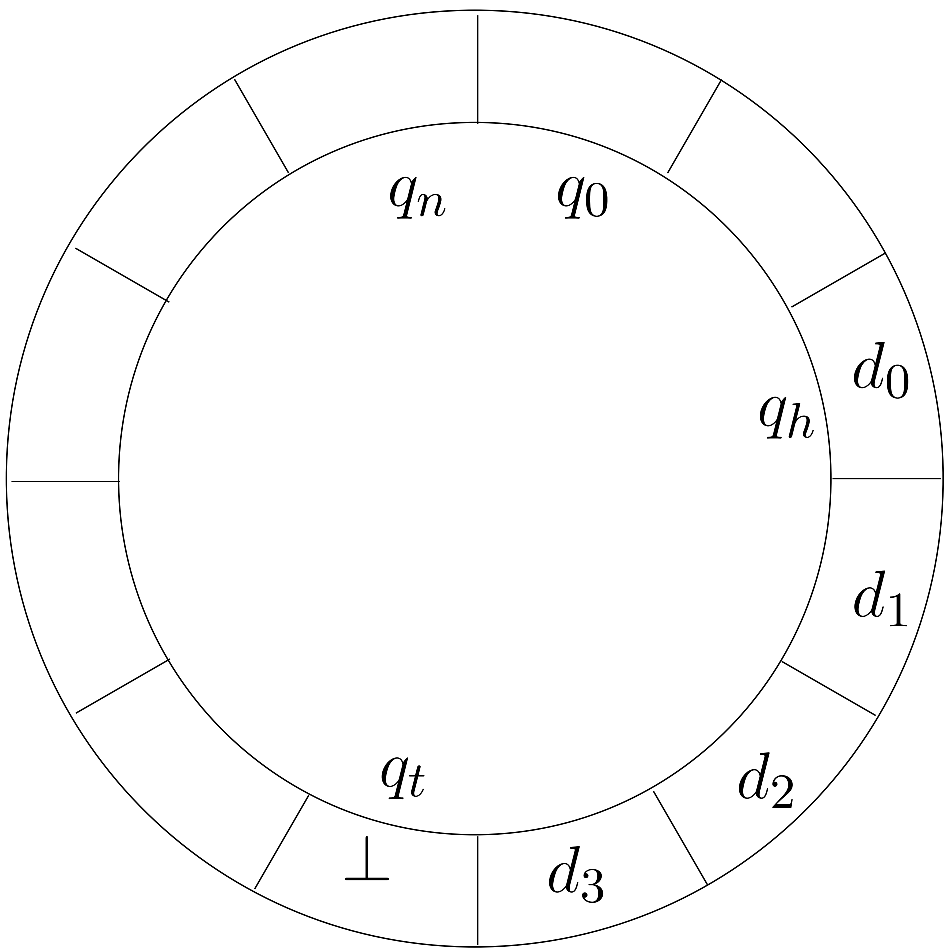
- No nosso exemplo, nós precisávamos que nossa “fila” fosse capaz de realizar as seguintes operações:
  - **inserir** *um elemento no* **fim** *da fila*
  - **recuperar e deletar** *o elemento no* **começo** *da fila*
  - *testar se a fila* **está vazia**
  - *encontrar o* **tamanho** *da fila*
- Uma vez que estas operações são repetidas com frequência, precisamos que elas sejam $O(1)$
- Poderíamos implementar filas utilizando:
  - arrays

    Precisaríamos ter que simular inserção/remoção; realocar array
  - listas

    Cada inserção envolve alocação de memória dinamicamente

Arrays circulares

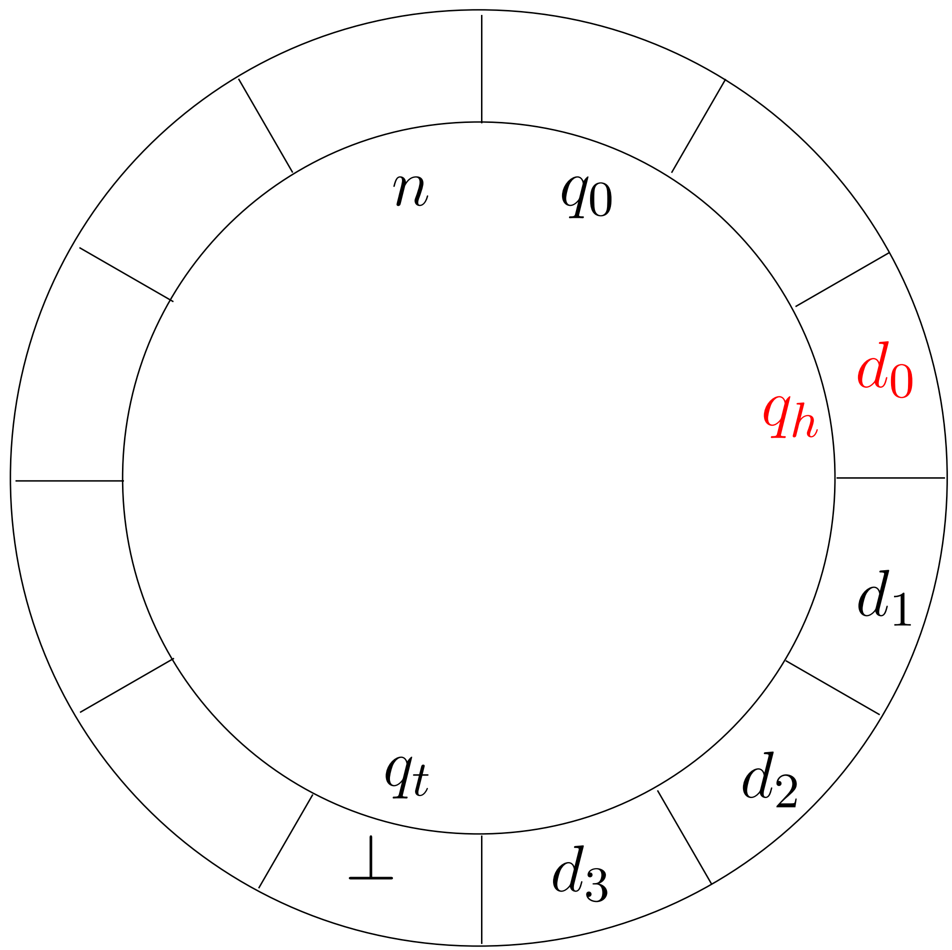
- Implementação usando um *array circular* q [Mehlhorn & Sanders]
- Usa aritmética modular (rápida)



- q: array $[0..n]$
- $q_h = d_0$ (índice do primeiro elemento da fila)
- $q_t = \perp$ (índice do último elemento da fila)
- a implementação de um array circular é simplesmente um array; mas o comportamento é diferente

Leitura do primeiro elemento da fila

```
first() {
  return q_h;
}
```

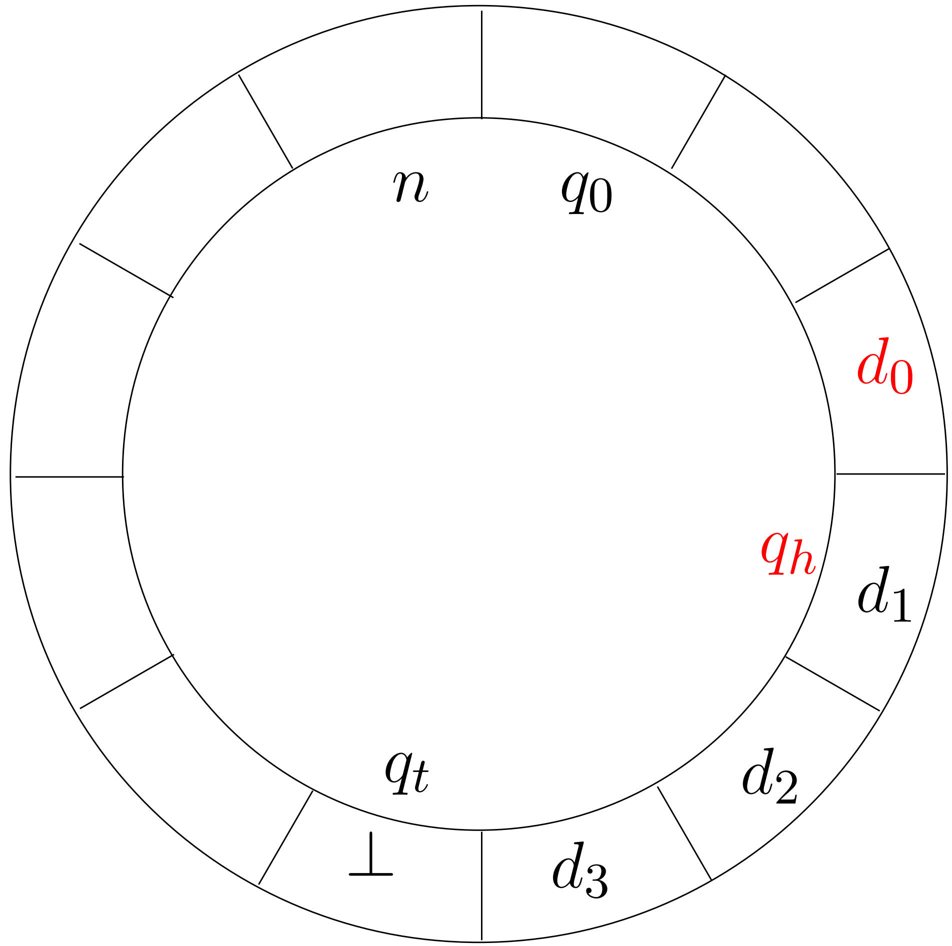


Deletar o elemento do começo da fila

$\texttt{popFront}()$ {

$p = q_h$;
$h = (h + 1) \bmod (n + 1)$;
return p;

}

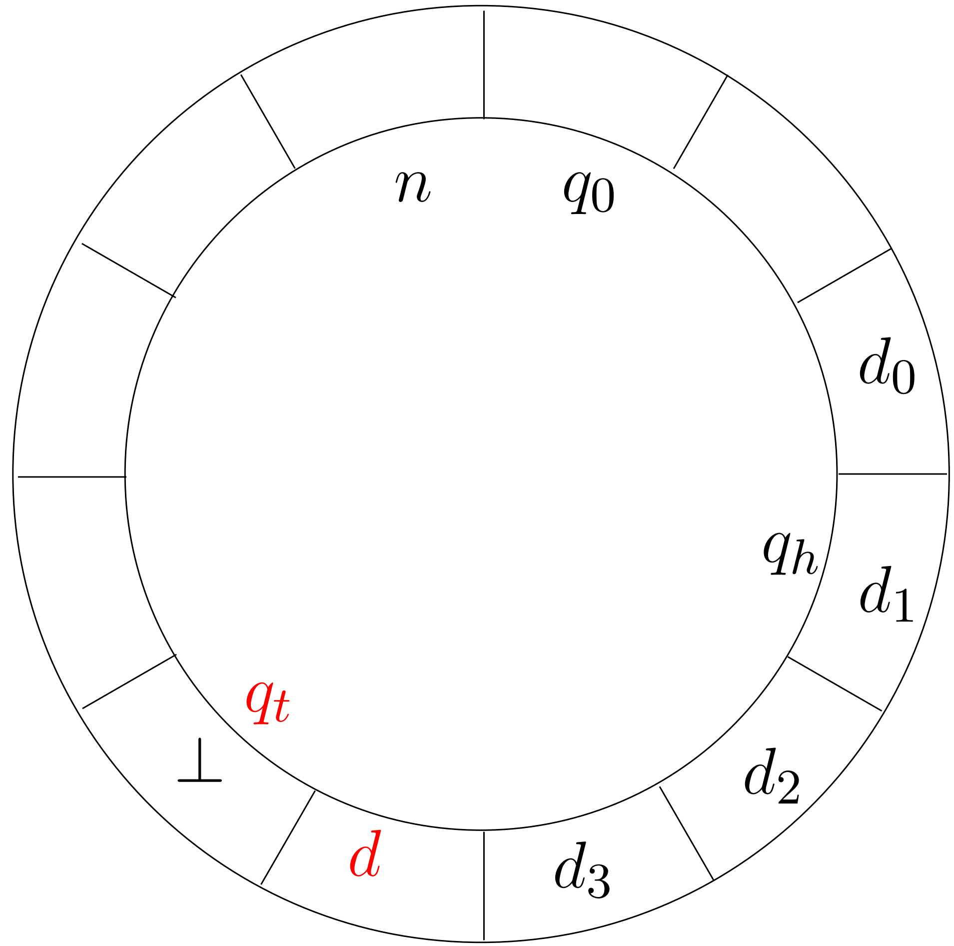


Inserir um elemento no fim da fila

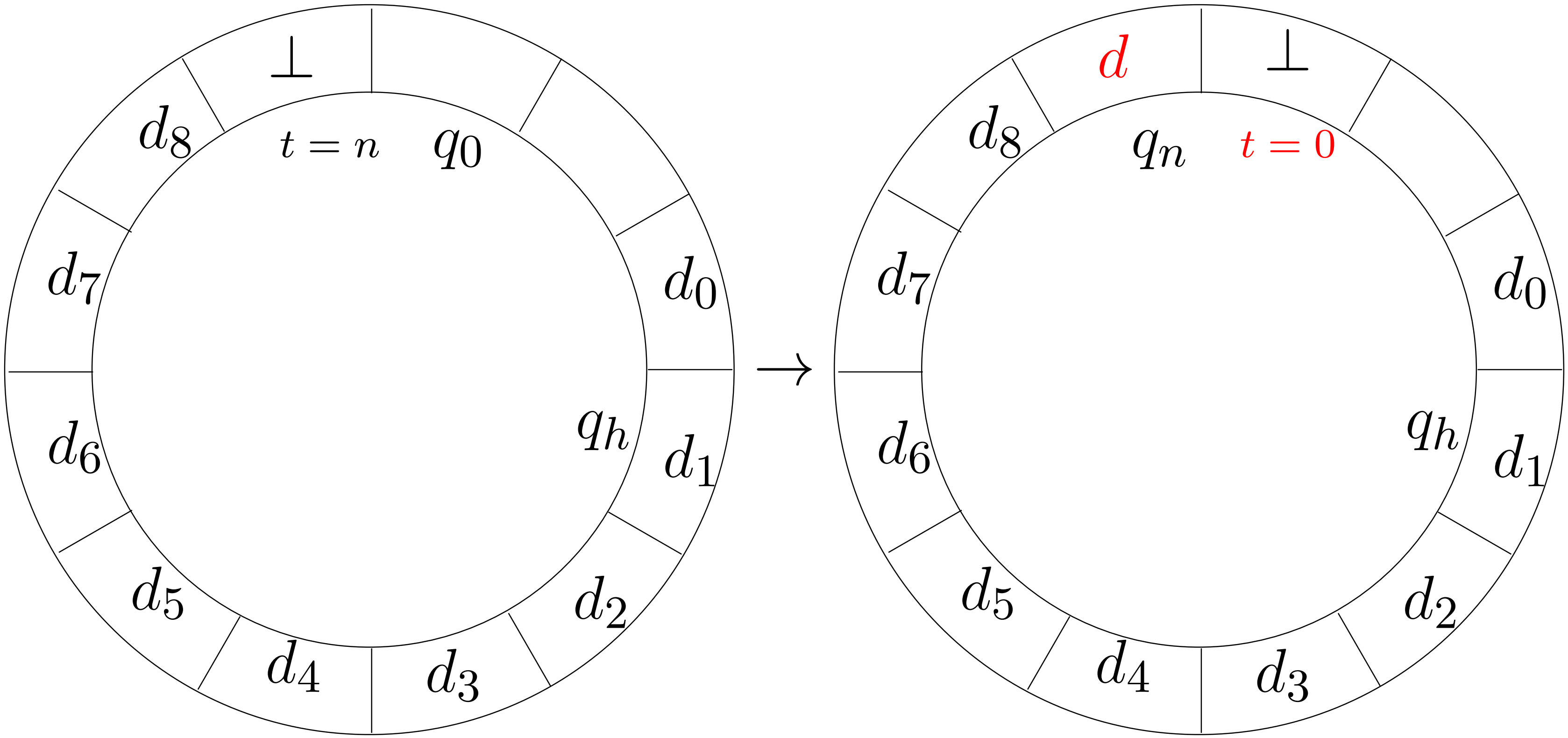
`pushBack`(d) {

  assert$($`size`$() < n)$
  $q_t = d$;
  $t = (t + 1) \bmod (n + 1)$;
  $q_t = \perp$;

}



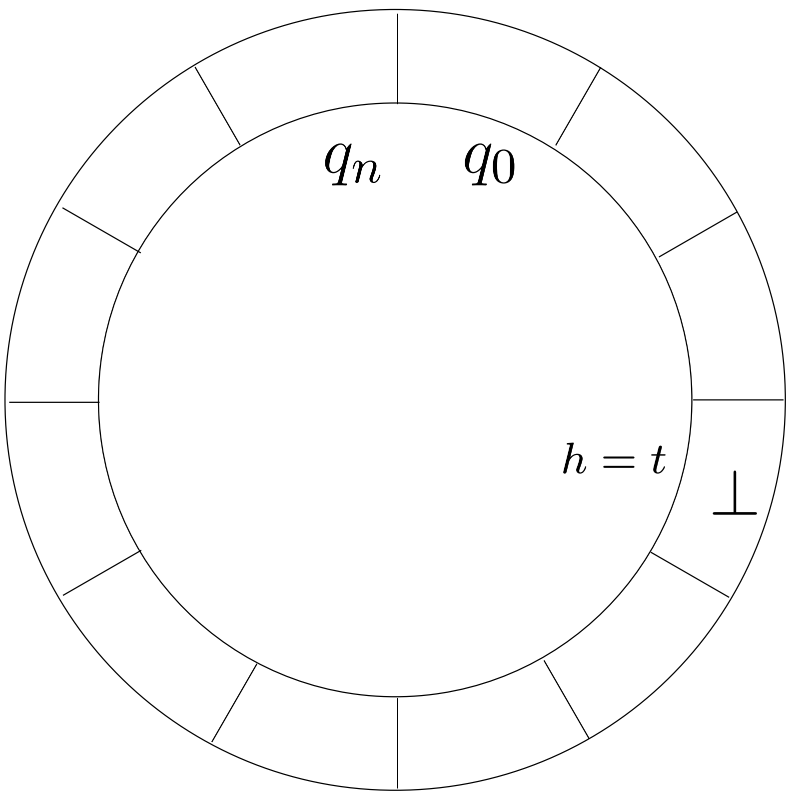
Inserir elemento no fim (caso $t = n$)



$t = (t+1) \bmod (n+1)$ com $t = n$ implica $t = 0$

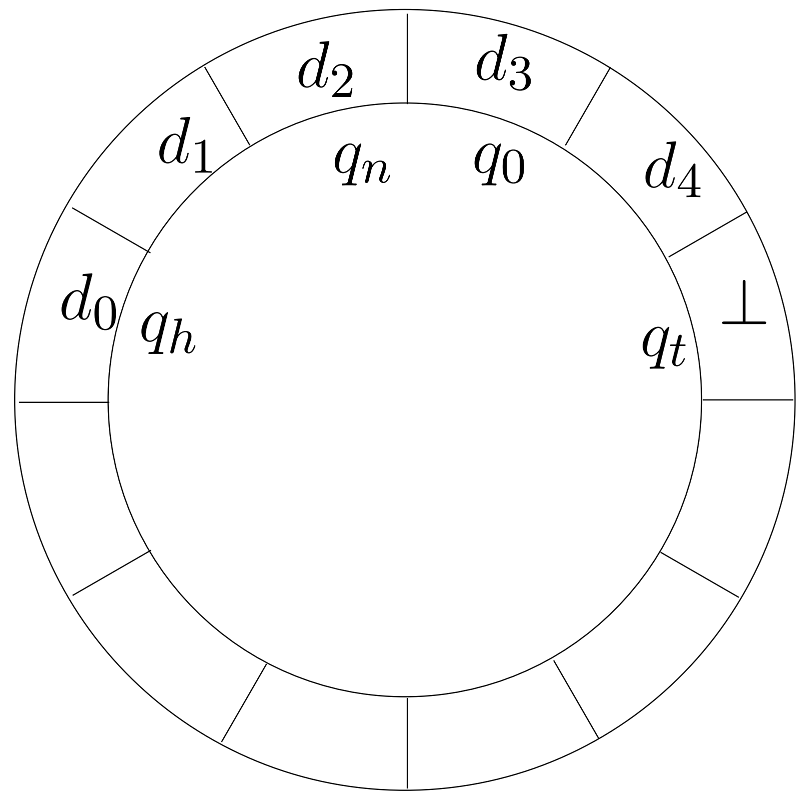
Tamanho e testa vazio

- `isEmpty()`: **if** $(t = h)$ **then return true; else return false;**



Tamanho e testa vazio

- `isEmpty`$()$: **if** $(t = h)$ **then return true; else return false;**
- `size`$()$: **return** $(t - h + n + 1) \bmod (n + 1)$;



Busca em Largura (BFS)

O algoritmo BFS

- **Input:** conjunto V, relação binária $\sim$ em V, e $s \neq t \in V$

$(Q, <) = \{s\}$; $L = \{s\}$;
while $Q \neq \varnothing$ **do**
  $u = \min_< Q$;
  $Q \leftarrow Q \setminus \{u\}$;
  for $v \in V$ $(v \sim u \wedge v \notin L)$ **do**
    if $v = t$ **then**
      return "t alcançado";
    end if
    $Q \leftarrow Q \cup \{v\}$, faça $v = \max_< Q$;
    $L \leftarrow L \cup \{v\}$;
  end for
end while
return "t não ancançável";

A ordem dos elementos em Q

- O conjunto ordenado Q é implementado com uma fila

A ordem dos elementos em Q

- O conjunto ordenado Q é implementado com uma fila
- Todo $v \in V$ entra em Q como o elemento *máximo* (i.e. o último)

A ordem dos elementos em Q

- O conjunto ordenado Q é implementado com uma fila
- Todo $v \in V$ entra em Q como o elemento *máximo* (i.e. o último)
- Nós apenas lemos (e removemos) o elemento *mínimo* de Q (i.e. o primeiro)

A ordem dos elementos em Q

- O conjunto ordenado Q é implementado com uma fila
- Todo $v \in V$ entra em Q como o elemento *máximo* (i.e. o último)
- Nós apenas lemos (e removemos) o elemento *mínimo* de Q (i.e. o primeiro)
- Os outros elementos de Q não são modificados

A ordem dos elementos em Q

- O conjunto ordenado Q é implementado com uma fila
- Todo $v \in V$ entra em Q como o elemento *máximo* (i.e. o último)
- Nós apenas lemos (e removemos) o elemento *mínimo* de Q (i.e. o primeiro)
- Os outros elementos de Q não são modificados
- A ordem relativa de uma subsequência consecutiva $u_1, \ldots, u_h$ de Q não é modificada

A ordem dos elementos em Q

- O conjunto ordenado Q é implementado com uma fila
- Todo $v \in V$ entra em Q como o elemento *máximo* (i.e. o último)
- Nós apenas lemos (e removemos) o elemento *mínimo* de Q (i.e. o primeiro)
- Os outros elementos de Q não são modificados
- A ordem relativa de uma subsequência consecutiva $u_1, \ldots, u_h$ de Q não é modificada
- Além disso, através do teste $v \notin L$ no Step 5, temos que:
  Thm. 1
  Nenhum elemento de V entra em Q mais de uma vez

Uma hierarquia de nós

- Considere uma função $\alpha : V \to \mathbb{N}$ definida como a seguir:

no Step 1, $\alpha(s) = 0$

no Step 9, $\alpha(v) = \alpha(u) + 1$

- Isto ordena os elementos de V pela distância a partir de s em termos de pares de relações
- E.g. Se $s \sim u$, então a distância de u a partir de s é 1 se $s \sim u \sim v$, a distância de v a partir de s é 2

O algoritmo BFS, de novo

$(Q,<) = \{s\}$; $L = \{s\}$;
$\alpha(s) = 0$;
while $Q \neq \varnothing$ **do**
  $u = \min_{<} Q$;
  $Q \leftarrow Q \setminus \{u\}$;
  for $v \in V$ $(v \sim u \wedge v \notin L)$ **do**
    $\alpha(v) = \alpha(u) + 1$;
    if $v = t$ **then**
      return "t alcançado";
    end if
    $Q \leftarrow Q \cup \{v\}$, set $v = \max_{<} Q$;
    $L \leftarrow L \cup \{v\}$;
  end for
end while
return "t não alcançável";

Resultados básicos

Temos os seguintes resultados (tente prová-los):

Thm. 2

Se $(s, v_1, \ldots, v_k)$ é um itinerário encontrado por BFS, $\alpha(v_k) = k$

Resultados básicos

Temos os seguintes resultados (tente prová-los):

Thm. 2

Se $(s, v_1, \ldots, v_k)$ é um itinerário encontrado por BFS, $\alpha(v_k) = k$

Thm. 3

Se $\alpha(u) < \alpha(v)$, então u entra em Q antes de v

Resultados básicos

Temos os seguintes resultados (tente prová-los):

Thm. 2

Se $(s, v_1, \ldots, v_k)$ é um itinerário encontrado por BFS, $\alpha(v_k) = k$

Thm. 3

Se $\alpha(u) < \alpha(v)$, então u entra em Q antes de v

Thm. 4

Nenhum itinerário encontrado por BFS possui elementos repetidos

Resultados básicos

Temos os seguintes resultados (tente prová-los):

Thm. 2

Se $(s, v_1, \ldots, v_k)$ é um itinerário encontrado por BFS, $\alpha(v_k) = k$

Thm. 3

Se $\alpha(u) < \alpha(v)$, então u entra em Q antes de v

Thm. 4

Nenhum itinerário encontrado por BFS possui elementos repetidos

Thm. 5

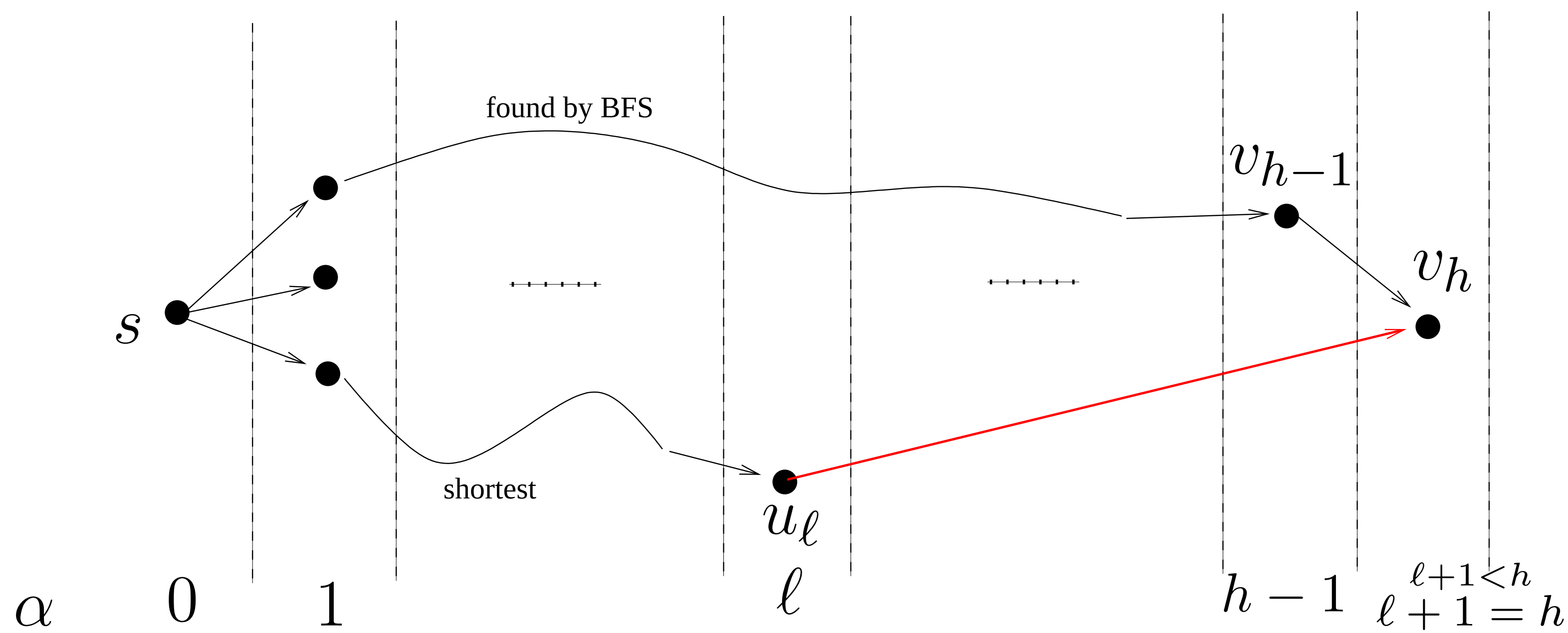
A função α é bem definida

Menor número de trocas

- O objetivo é provar que **BFS encontra um itinerário com o menor número de trocas**
- **Atenção**: o número de trocas em um itinerário é o mesmo que o número de nós do itinerário, portanto: Thm.

  BFS encontra o itinerário mais curto

Ideia da prova:



A prova

Thm.

BFS encontra o itinerário mais curto (em termos do número de nós)

Proof

Seja $R = (s, v_1, \ldots, v_k = t)$ o itinerário encontrado por BFS: e suponha que ele não seja o itinerário mais curto. Seja $h \leq k$ o menor índice tal que $R' = (s, \ldots, v_{h-1}, v_h)$ não é o caminho mais curto de $s \to v_h$. Pelo Thm. 2, $\alpha(v_h) = h$. Uma vez que este itinerário não é o mais curto, existe então outro itinerário $P = (s, u_1, \ldots, u_\ell, v_h)$ que é o mais curto: portanto necessariamente temos $\ell + 1 < h$, com $\ell < h$. Pelas propriedade do BFS $\alpha(u_\ell) \leq \ell$; além disso pelo Thm. 3 u_ℓ entra em Q antes de v_h, e então BFS encontra o itinerário P antes de R'. Temos então que $u_\ell \sim v_h$ resulta em $\alpha(v_h) \leq \ell + 1 < h = \alpha(v_h)$, contradição.

Todos os itinerários mais curtos

- Delete os Steps 8-10
- Todos os elementos em V entram e saem de Q
- Encontra os itinerários mais curtos a partir de s para todos os elementos de V

ATENÇÃO: BFS *não irá* encontrar os caminhos mais curtos em um grafo ponderado a não ser que todos os arcos tenham custo 1

E o mais rápido?

- Sempre que inserimos v no fim de Q, também armazenamos o tempo de chegada em v usando a informação do tempo de viagem contida na relação $u \sim v$

E o mais rápido?

- Sempre que inserimos v no fim de Q, também armazenamos o tempo de chegada em v usando a informação do tempo de viagem contida na relação $u \sim v$
- Obtemos o *tempo mínimo* τ' entre todos os elementos de Q

E o mais rápido?

- Sempre que inserimos v no fim de Q, também armazenamos o tempo de chegada em v usando a informação do tempo de viagem contida na relação $u \sim v$
- Obtemos o *tempo mínimo* τ' entre todos os elementos de Q
- Sempre que extraimos o nó t de Q, temos um possível itinerário $s \to t$; entre estes itinerários, guardamos o melhor tempo de chegada τ^* para t encontrado até então

E o mais rápido?

- Sempre que inserimos v no fim de Q, também armazenamos o tempo de chegada em v usando a informação do tempo de viagem contida na relação $u \sim v$
- Obtemos o *tempo mínimo* τ' entre todos os elementos de Q
- Sempre que extraimos o nó t de Q, temos um possível itinerário $s \to t$; entre estes itinerários, guardamos o melhor tempo de chegada τ^* para t encontrado até então
- Atualizamos τ^* sempre que encontramos um melhor itinerário $s \to t$

E o mais rápido?

- Sempre que inserimos v no fim de Q, também armazenamos o tempo de chegada em v usando a informação do tempo de viagem contida na relação $u \sim v$
- Obtemos o *tempo mínimo* τ' entre todos os elementos de Q
- Sempre que extraimos o nó t de Q, temos um possível itinerário $s \to t$; entre estes itinerários, guardamos o melhor tempo de chegada τ^* para t encontrado até então
- Atualizamos τ^* sempre que encontramos um melhor itinerário $s \to t$
- Assim que $\tau' \geq \tau^*$, sabemos que temos o itinerário mais rápido (por quê?)

Implementação

Uma implementação possível

L é inicializado com $\{s\}$;
Q é uma fila vazia;
$\alpha(s) = 0$;
Q.`pushBack`(s);
while $\neg(Q$.`isEmpty`$())$ **do**
  $u = Q$.`popFront`$()$;
  for $v \in V \; (v \sim u \wedge v \notin L)$ **do**
    $\alpha(v) = \alpha(u) + 1$
    if $v = t$ **then**
      return $\alpha(v)$;
    end if
    Q.`pushBack`(v);
    L.`pushBack`(v);
  end for
end while
return "t não alcançável";

$L = \{s\}$;
$(Q, <) = \varnothing$;
$\alpha(s) = 0$;
$(Q, <) = \{s\}$;
while $Q \neq \varnothing$ **do**
  $u = \min_< Q$; $Q \leftarrow Q \setminus \{u\}$;
  for $v \in V \; (v \sim u \wedge v \notin L)$ **do**
    $\alpha(v) = \alpha(u) + 1$
    if $v = t$ **then**
      return $\alpha(v)$;
    end if
    $Q \leftarrow Q \cup \{v\}$ $(v = \max_< Q)$;
    $L \leftarrow L \cup \{v\}$;
  end for
end while
return "t não alcançável";

Problemas

- Como você implementaria o Step 7?

$$\boxed{\textbf{for } v \in V \ (v \sim u \wedge v \notin L) \textbf{ do}}$$

Problemas

- Como você implementaria o Step 7?

$$\boxed{\textbf{for } v \in V \ (v \sim u \wedge v \notin L) \textbf{ do}}$$

- Looping em V: pior caso $O(|V|)$

Problemas

- Como você implementaria o Step 7?

$$\boxed{\textbf{for } v \in V \ (v \sim u \wedge v \notin L) \textbf{ do}}$$

- Looping em V: pior caso $O(|V|)$
- Para cada v, checar quando $v \sim u$

Problemas

- Como você implementaria o Step 7?

$$\boxed{\textbf{for } v \in V \ (v \sim u \wedge v \notin L) \textbf{ do}}$$

- Looping em V: pior caso $O(|V|)$
- Para cada v, checar quando $v \sim u$
- Podemos utilizar arrays denteados!

Problemas

- Como você implementaria o Step 7?

$$\boxed{\textbf{for } v \in V\ (v \sim u \wedge v \notin L) \textbf{ do}}$$

- Looping em V: pior caso $O(|V|)$
- Para cada v, checar quando $v \sim u$
- Podemos utilizar arrays denteados!
- Para cada v, checar quando $v \notin L$: pior caso $O(|V|)$ (quando a maioria dos nós de V são postos em L)

Problemas

- Como você implementaria o Step 7?

$$\boxed{\textbf{for } v \in V \ (v \sim u \wedge v \notin L) \textbf{ do}}$$

- Looping em V: pior caso $O(|V|)$
- Para cada v, checar quando $v \sim u$
- Podemos utilizar arrays denteados!
- Para cada v, checar quando $v \notin L$: pior caso $O(|V|)$ (quando a maioria dos nós de V são postos em L)
- Uma complexidade de pior caso de $O(|V|(1 + |V|)) = O(|V|^2)$

Repetição no loop mais externo: $O(|V|^3)$ ao todo: **ugh!**

Uma alternativa mais eficiente

- Inicializamos a função α de modo que $\alpha(u) = |V| + 1$ para todo $u \in V \setminus \{s\}$ antes do Step 5

Uma alternativa mais eficiente

- Inicializamos a função α de modo que $\alpha(u) = |V| + 1$ para todo $u \in V \setminus \{s\}$ antes do Step 5
- Repare que, por indução a partir de $\alpha(s) = 0$, sempre que $\alpha(v)$ é atualizado no Step 8 seu valor é sempre $\leq |V|$

Uma alternativa mais eficiente

- Inicializamos a função α de modo que $\alpha(u) = |V| + 1$ para todo $u \in V \setminus \{s\}$ antes do Step 5
- Repare que, por indução a partir de $\alpha(s) = 0$, sempre que $\alpha(v)$ é atualizado no Step 8 seu valor é sempre $\leq |V|$
- Uma vez que v é inserido em L depois de $\alpha(v)$ ser atualizado no Step 8, para cada $v \in V$ temos que $v \in L$ se e somente se $\alpha(v) \leq |V|$

Uma alternativa mais eficiente

- Inicializamos a função α de modo que $\alpha(u) = |V| + 1$ para todo $u \in V \setminus \{s\}$ antes do Step 5
- Repare que, por indução a partir de $\alpha(s) = 0$, sempre que $\alpha(v)$ é atualizado no Step 8 seu valor é sempre $\leq |V|$
- Uma vez que v é inserido em L depois de $\alpha(v)$ ser atualizado no Step 8, para cada $v \in V$ temos que $v \in L$ se e somente se $\alpha(v) \leq |V|$
- Modifique o loop no Step 7 como a seguir:

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

Uma alternativa mais eficiente

- Inicializamos a função α de modo que $\alpha(u) = |V| + 1$ para todo $u \in V \setminus \{s\}$ antes do Step 5
- Repare que, por indução a partir de $\alpha(s) = 0$, sempre que $\alpha(v)$ é atualizado no Step 8 seu valor é sempre $\leq |V|$
- Uma vez que v é inserido em L depois de $\alpha(v)$ ser atualizado no Step 8, para cada $v \in V$ temos que $v \in L$ se e somente se $\alpha(v) \leq |V|$
- Modifique o loop no Step 7 como a seguir:

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

- Agora a complexidade de pior caso é $O(|V|)$ (leitura de posição do vetor $O(1)$)

Uma alternativa mais eficiente

- Inicializamos a função α de modo que $\alpha(u) = |V| + 1$ para todo $u \in V \setminus \{s\}$ antes do Step 5
- Repare que, por indução a partir de $\alpha(s) = 0$, sempre que $\alpha(v)$ é atualizado no Step 8 seu valor é sempre $\leq |V|$
- Uma vez que v é inserido em L depois de $\alpha(v)$ ser atualizado no Step 8, para cada $v \in V$ temos que $v \in L$ se e somente se $\alpha(v) \leq |V|$
- Modifique o loop no Step 7 como a seguir:

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

- Agora a complexidade de pior caso é $O(|V|)$ (leitura de posição do vetor $O(1)$)

Desta forma, temos $O(|V|^2)$ ao todo

Análise da complexidade do pior caso

- O loop interno

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

apenas executa para as relações $(u, v) \in E$, onde E é conjunto de arestas do grafo.

Análise da complexidade do pior caso

- O loop interno

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

  apenas executa para as relações $(u, v) \in E$, onde E é conjunto de arestas do grafo.

- Devido ao fato de u nunca ser considerado mais de uma vez, segue que os pares (u, v) são jamais considerados mais de uma vez nas iterações de ambos os loops.

Análise da complexidade do pior caso

- O loop interno

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

apenas executa para as relações $(u, v) \in E$, onde E é conjunto de arestas do grafo.

- Devido ao fato de u nunca ser considerado mais de uma vez, segue que os pares (u, v) são jamais considerados mais de uma vez nas iterações de ambos os loops.
- No pior caso, a instrução Q.`pushBack`(v) pode ser repetida tantas vezes quanto existem pares de relação $\sim$ (= $|E|$: número de arcos do grafo).

Análise da complexidade do pior caso

- O loop interno

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

  apenas executa para as relações $(u, v) \in E$, onde E é conjunto de arestas do grafo.

- Devido ao fato de u nunca ser considerado mais de uma vez, segue que os pares (u, v) são jamais considerados mais de uma vez nas iterações de ambos os loops.

- No pior caso, a instrução Q.pushBack(v) pode ser repetida tantas vezes quanto existem pares de relação $\sim$ (= $|E|$: número de arcos do grafo).

- Além disso, executamos o corpo do loop mais externo no máximo $|V|$ vezes

Análise da complexidade do pior caso

- O loop interno

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

  apenas executa para as relações $(u, v) \in E$, onde E é conjunto de arestas do grafo.
- Devido ao fato de u nunca ser considerado mais de uma vez, segue que os pares (u, v) são jamais considerados mais de uma vez nas iterações de ambos os loops.
- No pior caso, a instrução Q.`pushBack`(v) pode ser repetida tantas vezes quanto existem pares de relação $\sim$ (= $|E|$: número de arcos do grafo).
- Além disso, executamos o corpo do loop mais externo no máximo $|V|$ vezes
- Assintóticamente, não podemos comparar $|V|, |E|$: depende do grafo

Análise da complexidade do pior caso

- O loop interno

$$\textbf{for } v \in V\ (v \sim u \wedge \alpha(v) = |V| + 1)\ \textbf{do}$$

apenas executa para as relações $(u, v) \in E$, onde E é conjunto de arestas do grafo.

- Devido ao fato de u nunca ser considerado mais de uma vez, segue que os pares (u, v) são jamais considerados mais de uma vez nas iterações de ambos os loops.
- No pior caso, a instrução Q.pushBack(v) pode ser repetida tantas vezes quanto existem pares de relação $\sim$ (= $|E|$: número de arcos do grafo).
- Além disso, executamos o corpo do loop mais externo no máximo $|V|$ vezes
- Assintóticamente, não podemos comparar $|V|, |E|$: depende do grafo
- $\Rightarrow$ A complexidade de pior caso do BFS é portanto $O(|V| + |E|)$

Fim do módulo 3